NixOS

main administration’s commands and concepts

Table of contents

NixOS concepts....2
Introduction....2
The big concepts....2
Vocabulary words to stick with NixOS concept prac tice....3
The place NixOS looks at, you should know about....5
Commands to manage NixOS....6
Introduction to this list....6
The table list....6
Best command to use at first place to manage NixOS....6
nix.conf file....8
Introduction....8
variables content names....8

NixOS concepts

Introduction

NixOS use a functional system configuration's concept to rich administrative tasks. And obviously with modularity depend of system wise result, or user specific target or specific own application to install.
It gives users and root user the ability to manage system and own works the easy way by his own tools and concept. Learn NixOS is not natural, but when you get the idea, it will have a quick learning curve for a good power to administrate your Linux. NixOs is difficult to break as long as you trust his tools only.

The big concepts

Nixos has own tools to manage system
Nixos install applications in a unique path and link binaries where they have to be
Nixos is modular and structured
Nixos is an equivalent for administartion system model to functionnal programmation langage can be
Nixos is offering facilities to manage applications
Nixos can install many versions of the same application you will link with other or switch between
Nixos has a kind of versioning system and can rollback on other old administrative status profile
Nixos is safe to install and reproductible because of his functionnal way to config and install things
Nixos has channels of version systems it can be, the way you will add, remov and switch on

Own big lexical fields in relation

Usually, the big concepts lexical fields with NixOS to know about to help to understand NixOS are :

1. Channel
2. Profile / Generation
3. Store / Derivations /Closure
4. NIX Expression

profile

The NixOS step at a specific generation of a given existing user. A user has many profiles step, each one is a generation, they are all of them symbolic links. The current profile is the symlink to the current user environment of a user, e.g., `/nix/var/nix/profiles/default`.

store

The location in the file system where NixOS store objects (like applications and libraries) live. Typically `/nix/store`.

path

The location in the file system of a store object, i.e., an immediate child of the Nix store directory.

object

A file that is an immediate child of the Nix store directory. These can be regular files, but also entire directory trees. Store objects can be sources (objects copied from outside of the store), derivation outputs (objects produced by running a build action), or derivations (files describing a build action).

generation

Each generation define a specific profile state of the NixOS system for a given user you can move on. Each generation is a time own user logged system configuration step. You can then easily rollback.

derivation

It can be describe by a kind of confined package existing on the store. Then, a derivation can be link with an other one derivation it needs to run with. All the result of derivations, what ever linked they are is inside the store place when they finish to be generate in the system.
Derivations are typically specified in Nix expressions using the `derivation` primitive. These are translated into low-level *store derivations* (implicitly by **nix-env** and **nix-build**, or explicitly by **nix-instantiate**).

deriver

Describe the application sused to build a deriver (not depezndencies or parent, but specificaly the tools applications required to build the application... this derivation).

substitute

A substitute is a command invocation stored in the Nix database that describes how to build a store object, bypassing the normal build mechanism (i.e., derivations). Typically, the substitute builds the store object by downloading a pre-built version of the store object from some server.

purity

Is a special derivation with the assumption that equal Nix derivations at run will always produce the same output. This cannot be guaranteed in general (e.g., a builder can rely on external inputs such as the network or the system time) but the Nix model assumes it.

Nix expression

A high-level description of software packages and compositions thereof. Deploying software using Nix entails writing Nix expressions for your packages. Nix expressions are translated to derivations that are stored in the Nix store. These derivations can then be built.

reference

A store path `P` is said to have a reference to a store path `Q` if the store object at `P` contains the path `Q` somewhere. The *references* of a store path are the set of store paths to which it has a reference.
A derivation can reference other derivations and sources (but not output paths), whereas an output path only references other output paths.

reachable

A store path `Q` is reachable from another store path `P` if `Q` is in the closure of the references relation.

closure

The closure is a relational link of a derivation (dependency) or his runtime dependency. For correct deployment it is necessary to deploy whole closures, since otherwise at runtime files could be missing.

output path

A store path produced by a derivation.

validity

A store path is considered *valid* if it exists in the file system, is listed in the Nix database as being valid, and if all paths in its closure are also valid.

user environment

An automatically generated store object that consists of a set of symlinks to “active” applications, i.e., other store paths. These are generated automatically by **nix-env**. See Chapter 10, *Profiles*.

NAR

A *N*ix *AR*chive. This is a serialisation of a path in the Nix store. It can contain regular files, directories and symbolic links. NARs are generated and unpacked using **nix-store --dump** and **nix-store --restore**.

The place NixOS looks at, you should know about

- /nix/store / contain applications containers « store »
- Specific wise configuration system files you have to setup to apply on system are :
  - /etc/nixos/configuration.nix
  - /etc/nixos/hardware-configuration.nix
- */run/*current-system/sw/ path to find any links of applications system widely installed
- */nix/*var/nix/profiles/<user>/ contain users own usefull links to applications or whatever things installed for this <user>
- .nix files are the one to use to config any NixOS requierement files to build anything
- ~/.nix-profile/ is the default place to find user links to profiles of generations

Commands to manage NixOS

Introduction to this list

Which command should be learn at first place to start with NixOS to rich « best practices ».
Lower priority number target on the usual best choice first to manage NixOS.
Priority 0 is a first place to go, but not the most usual one...
At a next step learning point, you would probably see this classification as unusual or far from your nixos own use case. It is definitely not a kind of low to follow this indication, but it would like to be helpfull for you to target quickly the current manager commands to use with NixOS at the very first time. All command priority up to 3 should be avoid from beginner's users and are used by other nixos-* commands.

The table list

nixos-help	open own prefered browser to read NixOS afficial Manual page	1
nixos-rebuild	re-build NixOS system wise from configuration.nix file and options to manage this full system configuration to happen	1
nix-collect-garbage	will wash NixOs store concerned by unused packages from your store	1
nix-info	Show NixOS info in relation with system and user	2
nixos-version	show current NixOS version (and with option, the commit in relation from git repo)	2
nixos-channel	manage « channel » NixOS concept repository	2
nixos-generate-config	will generate first place configurations template files to next go for build NixOS	2
nix-shell	manage packages for user concerned area	3
nixos-option	show value concerned from entry option.name and will discribe it from NixOS configuration.nix file	3
nix-store	manipulate NixOS target store to do action with (used by other script nixos-*)	4
nix-prefetch-url	will have to download and store from an url and return the hash value of a package	5
nix-instantiate	will return instance of store derivation depend from options	5
nixos-install	install NixOS bootloader	6
nixos-enter	to run a command inside a chrooted NixOS environement created from nixos-install	6
nixos-container	Manage NixOS containers	6
nix-build	build NixOS expression and give back a results links files (for each expression to build from a config file from path), it is the old way to manage derivations	7
nix-copy-closure	to copy through ssh closure of NixOS remote machines sowftware(s) target from	8
nixos-build-vms	to build virtual machine network from network nixos config	8
nix-hash	checksum result for a path to be used for ghet an idea of nixos store files and directories names to be	8

Best command to use at first place to manage NixOS

System wise administration / configurations

- on first install, generate defaults configuration files with nix-generate-config command
- add configuration setups inside */etc*/nixos/configuration.nix and save
- run command nixos-rebuild switch to apply config now

System and users general commands to use

- The command nix-collect-garbage can be used to remove unused derviations from store (you can use it after to full update channel and derivations)
- nix-channel command is used to manage channels (user or system wise when used as root)
  - --list option will show your channels
  - --add <url> [<name>] will ad a channel from url with <name> tag linked with
  - --remove <name> will remove channel tag <name> nammed
  - --update [<names> ...] will update tag listed channels list of each <name> and will generate then a new generation the current profile will go on.
  - --rollback [<generation ID>] will rollback to the previous generation or the one provide with <generation> argument

User system administrations tasks

- NixOS can have specific application for user to manage things by using the command : nix-env
  - user can manage « profiles » of « generations » with options :
    - --list-generations to list generations of profiles
    - -G <profile ID (int)>
  - user can query for appllications with options with optionaly a regex target :
    - -q [<regex-app-name>] to check local user applications installed
    - -qa [<regex-app-name>] to check online aviable applications (on current used channel)
    - -q[a]s [<regex-app-name>] the s flag will add in first column the status of derivations (IPS flags to read on each line)
  - user can install applications with option :
    - -i <regex-app-name> to install an application target with its closures on store
  - user can remove application witrh option :
    - -e [<regex-app-name>] to erase application derivations from store
  - user can update applications (you should first update the current channel) with :
    - -u [<regex-app-name>] to update targeted application derivations
- NixOS can handle some configuaration file(s) to get a virtual shell handle on your local path at call time. This virtual env will be usefull to build your own application derivation (this file named nix.conf). This file can live inside :
  - `$XDG_CONFIG_HOME/nix/nix.conf`
  - `~/.config/nix/nix.conf`
  - `at XDG_CONFIG_HOME environment variable path`.

NIX.CONF FILE

Introduction

Nix reads settings from two configuration files:

- The system-wide configuration file *sysconfdir*`/nix/nix.conf` (i.e. `/etc/nix/nix.conf` on most systems), or `$NIX_CONF_DIR/nix.conf` if `NIX_CONF_DIR` is set.
- The user configuration file `$XDG_CONFIG_HOME/nix/nix.conf`, or `~/.config/nix/nix.conf` if `XDG_CONFIG_HOME` is not set.

The configuration files consist of *`name = value`* pairs, one per line. Other files can be included with a line like `include` *`path`*, where *`path`* is interpreted relative to the current conf file and a missing file is an error unless `!include` is used instead. Comments start with a # character

variables content names

allowed-uris

A list of URI prefixes to which access is allowed in restricted evaluation mode. For example, when set to `https://github.com/NixOS`, builtin functions such as `fetchGit` are allowed to access `https://github.com/NixOS/patchelf.git`.

allow-import-from-derivation

By default, Nix allows you to `import` from a derivation, allowing building at evaluation time. With this option set to false, Nix will throw an error when evaluating an expression that uses this feature, allowing users to ensure their evaluation will not require any builds to take place.

allow-new-privileges

(Linux-specific.) By default, builders on Linux cannot acquire new privileges by calling setuid/setgid programs or programs that have file capabilities. For example, programs such as **sudo** or **ping** will fail. (Note that in sandbox builds, no such programs are available unless you bind-mount them into the sandbox via the `sandbox-paths` option.) You can allow the use of such programs by enabling this option. This is impure and usually undesirable, but may be useful in certain scenarios (e.g. to spin up containers or set up userspace network interfaces in tests).

allowed-users

A list of names of users (separated by whitespace) that are allowed to connect to the Nix daemon. As with the `trusted-users` option, you can specify groups by prefixing them with `@`. Also, you can allow all users by specifying `*`. The default is `*`. Note that trusted users are always allowed to connect.

auto-optimise-store

If set to `true`, Nix automatically detects files in the store that have identical contents, and replaces them with hard links to a single copy. This saves disk space. If set to `false` (the default), you can still run **nix-store --optimise** to get rid of duplicate files.

`builders`

A list of machines on which to perform builds. See Chapter 16, *Remote Builds* for details.

builders-use-substitutes

If set to `true`, Nix will instruct remote build machines to use their own binary substitutes if available. In practical terms, this means that remote hosts will fetch as many build dependencies as possible from their own substitutes (e.g, from `cache.nixos.org`), instead of waiting for this host to upload them all. This can drastically reduce build times if the network connection between this computer and the remote build host is slow. Defaults to `false`.

build-users-group

This options specifies the Unix group containing the Nix build user accounts. In multi-user Nix installations, builds should not be performed by the Nix account since that would allow users to arbitrarily modify the Nix store and database by supplying specially crafted builders; and they cannot be performed by the calling user since that would allow him/her to influence the build result.
Therefore, if this option is non-empty and specifies a valid group, builds will be performed under the user accounts that are a member of the group specified here (as listed in `/etc/group`). Those user accounts should not be used for any other purpose!
Nix will never run two builds under the same user account at the same time. This is to prevent an obvious security hole: a malicious user writing a Nix expression that modifies the build result of a legitimate Nix expression being built by another user. Therefore it is good to have as many Nix build user accounts as you can spare. (Remember: uids are cheap.)
The build users should have permission to create files in the Nix store, but not delete them. Therefore, `/nix/store` should be owned by the Nix account, its group should be the group specified here, and its mode should be `1775`.
If the build users group is empty, builds will be performed under the uid of the Nix process (that is, the uid of the caller if `NIX_REMOTE` is empty, the uid under which the Nix daemon runs if `NIX_REMOTE` is `daemon`). Obviously, this should not be used in multi-user settings with untrusted users.
`compress-build-log`
If set to `true` (the default), build logs written to `/nix/var/log/nix/drvs` will be compressed on the fly using bzip2. Otherwise, they will not be compressed.

connect-timeout

The timeout (in seconds) for establishing connections in the binary cache substituter. It corresponds to **curl**'s `--connect-timeout` option.

cores

Sets the value of the `NIX_BUILD_CORES` environment variable in the invocation of builders. Builders can use this variable at their discretion to control the maximum amount of parallelism. For instance, in Nixpkgs, if the derivation attribute `enableParallelBuilding` is set to `true`, the builder passes the `-jN` flag to GNU Make. It can be overridden using the `--cores` command line switch and defaults to `1`. The value `0` means that the builder should use all available CPU cores in the system.
See also Chapter 17, *Tuning Cores and Jobs*.

diff-hook

Absolute path to an executable capable of diffing build results. The hook executes if `run-diff-hook` is true, and the output of a build is known to not be the same. This program is not executed to determine if two results are the same.
The diff hook is executed by the same user and group who ran the build. However, the diff hook does not have write access to the store path just built.
The diff hook program receives three parameters:

1. A path to the previous build's results
2. A path to the current build's results
3. The path to the build's derivation
4. The path to the build's scratch directory. This directory will exist only if the build was run with `--keep-failed`.

The stderr and stdout output from the diff hook will not be displayed to the user. Instead, it will print to the nix-daemon's log.
When using the Nix daemon, `diff-hook` must be set in the `nix.conf` configuration file, and cannot be passed at the command line.

enforce-determinism

See `repeat`.

extra-sandbox-paths

A list of additional paths appended to `sandbox-paths`. Useful if you want to extend its default value.

extra-platforms

Platforms other than the native one which this machine is capable of building for. This can be useful for supporting additional architectures on compatible machines: i686-linux can be built on x86_64-linux machines (and the default for this setting reflects this); armv7 is backwards-compatible with armv6 and armv5tel; some aarch64 machines can also natively run 32-bit ARM code; and qemu-user may be used to support non-native platforms (though this may be slow and buggy). Most values for this are not enabled by default because build systems will often misdetect the target platform and generate incompatible code, so you may wish to cross-check the results of using this option against proper natively-built versions of your derivations.

extra-substituters

Additional binary caches appended to those specified in `substituters`. When used by unprivileged users, untrusted substituters (i.e. those not listed in `trusted-substituters`) are silently ignored.

fallback

If set to `true`, Nix will fall back to building from source if a binary substitute fails. This is equivalent to the `--fallback` flag. The default is `false`.

fsync-metadata

If set to `true`, changes to the Nix store metadata (in `/nix/var/nix/db`) are synchronously flushed to disk. This improves robustness in case of system crashes, but reduces performance. The default is `true`.

hashed-mirrors

A list of web servers used by `builtins.fetchurl` to obtain files by hash. The default is `http://tarballs.nixos.org/`. Given a hash type *`ht`* and a base-16 hash *h*, Nix will try to download the file from `hashed-mirror/`*`ht`*`/`*h*. This allows files to be downloaded even if they have disappeared from their original URI. For example, given the default mirror `http://tarballs.nixos.org/`, when building the derivation
builtins.fetchurl {
  url = https://example.org/foo-1.2.3.tar.xz;
  sha256 = "2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae";
}
Nix will attempt to download this file from
`http://tarballs.nixos.org/sha256/2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae`
first. If it is not available there, if will try the original URI.

http-connections

The maximum number of parallel TCP connections used to fetch files from binary caches and by other downloads. It defaults to 25. 0 means no limit.

keep-build-log

If set to `true` (the default), Nix will write the build log of a derivation (i.e. the standard output and error of its builder) to the directory `/nix/var/log/nix/drvs`. The build log can be retrieved using the command **nix-store -l** ***`path`***.

keep-derivations

If `true` (default), the garbage collector will keep the derivations from which non-garbage store paths were built. If `false`, they will be deleted unless explicitly registered as a root (or reachable from other roots).
Keeping derivation around is useful for querying and traceability (e.g., it allows you to ask with what dependencies or options a store path was built), so by default this option is on. Turn it off to save a bit of disk space (or a lot if `keep-outputs` is also turned on).

keep-env-derivations

If `false` (default), derivations are not stored in Nix user environments. That is, the derivations of any build-time-only dependencies may be garbage-collected.

If `true`, when you add a Nix derivation to a user environment, the path of the derivation is stored in the user environment. Thus, the derivation will not be garbage-collected until the user environment generation is deleted (**nix-env --delete-generations**). To prevent build-time-only dependencies from being collected, you should also turn on `keep-outputs`.
The difference between this option and `keep-derivations` is that this one is "sticky": it applies to any user environment created while this option was enabled, while `keep-derivations` only applies at the moment the garbage collector is run.

keep-outputs

If `true`, the garbage collector will keep the outputs of non-garbage derivations. If `false` (default), outputs will be deleted unless they are GC roots themselves (or reachable from other roots).
In general, outputs must be registered as roots separately. However, even if the output of a derivation is registered as a root, the collector will still delete store paths that are used only at build time (e.g., the C compiler, or source tarballs downloaded from the network). To prevent it from doing so, set this option to `true`.

max-build-log-size

This option defines the maximum number of bytes that a builder can write to its stdout/stderr. If the builder exceeds this limit, it's killed. A value of `0` (the default) means that there is no limit.

max-free

When a garbage collection is triggered by the `min-free` option, it stops as soon as `max-free` bytes are available. The default is infinity (i.e. delete all garbage).

max-jobs

This option defines the maximum number of jobs that Nix will try to build in parallel. The default is `1`. The special value `auto` causes Nix to use the number of CPUs in your system. `0` is useful when using remote builders to prevent any local builds (except for `preferLocalBuild` derivation attribute which executes locally regardless). It can be overridden using the `--max-jobs` (`-j`) command line switch.
See also Chapter 17, *Tuning Cores and Jobs*.

max-silent-time

This option defines the maximum number of seconds that a builder can go without producing any data on standard output or standard error. This is useful (for instance in an automated build system) to catch builds that are stuck in an infinite loop, or to catch remote builds that are hanging due to network problems. It can be overridden using the `--max-silent-time` command line switch.
The value `0` means that there is no timeout. This is also the default.

min-free

When free disk space in `/nix/store` drops below `min-free` during a build, Nix performs a garbage-collection until `max-free` bytes are available or there is no more garbage. A value of `0` (the default) disables this feature.

narinfo-cache-negative-ttl

The TTL in seconds for negative lookups. If a store path is queried from a substituter but was not found, there will be a negative lookup cached in the local disk cache database for the specified duration.

narinfo-cache-positive-ttl

The TTL in seconds for positive lookups. If a store path is queried from a substituter, the result of the query will be cached in the local disk cache database including some of the NAR metadata. The default TTL is a month, setting a shorter TTL for positive lookups can be useful for binary caches that have frequent garbage collection, in which case having a more frequent cache invalidation would prevent trying to pull the path again and failing with a hash mismatch if the build isn't reproducible.

netrc-file

If set to an absolute path to a `netrc` file, Nix will use the HTTP authentication credentials in this file when trying to download from a remote host through HTTP or HTTPS. Defaults to `$NIX_CONF_DIR/netrc`.
The `netrc` file consists of a list of accounts in the following format:
machine *`my-machine`*

login *my-username*
password *my-password*
For the exact syntax, see the curl documentation.
Note: This must be an absolute path, and ~ is not resolved. For example, `~/.netrc` won't resolve to your home directory's `.netrc`.

plugin-files

A list of plugin files to be loaded by Nix. Each of these files will be dlopened by Nix, allowing them to affect execution through static initialization. In particular, these plugins may construct static instances of RegisterPrimOp to add new primops or constants to the expression language, RegisterStoreImplementation to add new store implementations, RegisterCommand to add new subcommands to the `nix` command, and RegisterSetting to add new nix config settings. See the constructors for those types for more details.
Since these files are loaded into the same address space as Nix itself, they must be DSOs compatible with the instance of Nix running at the time (i.e. compiled against the same headers, not linked to any incompatible libraries). They should not be linked to any Nix libs directly, as those will be available already at load time.
If an entry in the list is a directory, all files in the directory are loaded as plugins (non-recursively).

pre-build-hook

If set, the path to a program that can set extra derivation-specific settings for this system. This is used for settings that can't be captured by the derivation model itself and are too variable between different versions of the same system to be hard-coded into nix.
The hook is passed the derivation path and, if sandboxes are enabled, the sandbox directory. It can then modify the sandbox and send a series of commands to modify various settings to stdout. The currently recognized commands are:

extra-sandbox-paths

Pass a list of files and directories to be included in the sandbox for this build. One entry per line, terminated by an empty line. Entries have the same format as `sandbox-paths`.

post-build-hook

Optional. The path to a program to execute after each build.
This option is only settable in the global `nix.conf`, or on the command line by trusted users.
When using the nix-daemon, the daemon executes the hook as `root`. If the nix-daemon is not involved, the hook runs as the user executing the nix-build.

- The hook executes after an evaluation-time build.
- The hook does not execute on substituted paths.
- The hook's output always goes to the user's terminal.
- If the hook fails, the build succeeds but no further builds execute.
- The hook executes synchronously, and blocks other builds from progressing while it runs.

The program executes with no arguments. The program's environment contains the following environment variables:

DRV_PATH

The derivation for the built paths.
Example: `/nix/store/5nihn1a7pa8b25l9zafqaqibznlvvp3f-bash-4.4-p23.drv`

OUT_PATHS

Output paths of the built derivation, separated by a space character.
Example: `/nix/store/zf5lbh336mnzf1nlswdn11g4n2m8zh3g-bash-4.4-p23-dev`
`/nix/store/rjxwxwv1fpn9wa2x5ssk5phzwlcv4mna-bash-4.4-p23-doc`
`/nix/store/6bqvbzjkcp9695dq0dpl5y43nvy37pq1-bash-4.4-p23-info`
`/nix/store/r7fng3kk3vlpdlh2idnrbn37vh4imlj2-bash-4.4-p23-man`
`/nix/store/xfghy8ixrhz3kyy6p724iv3cxji088dx-bash-4.4-p23`.
See Chapter 19, *Using the `post-build-hook`* for an example implementation.

repeat

How many times to repeat builds to check whether they are deterministic. The default value is 0. If the value is non-zero, every build is repeated the specified number of times. If the contents of any of the runs differs from the previous ones and enforce-determinism is true, the build is rejected and the resulting store paths are not registered as "valid" in Nix's database.

require-sigs

If set to true (the default), any non-content-addressed path added or copied to the Nix store (e.g. when substituting from a binary cache) must have a valid signature, that is, be signed using one of the keys listed in trusted-public-keys or secret-key-files. Set to false to disable signature checking.

restrict-eval

If set to true, the Nix evaluator will not allow access to any files outside of the Nix search path (as set via the NIX_PATH environment variable or the -I option), or to URIs outside of allowed-uri. The default is false.

run-diff-hook

If true, enable the execution of diff-hook.
When using the Nix daemon, run-diff-hook must be set in the nix.conf configuration file, and cannot be passed at the command line.

sandbox

If set to true, builds will be performed in a *sandboxed environment*, i.e., they're isolated from the normal file system hierarchy and will only see their dependencies in the Nix store, the temporary build directory, private versions of /proc, /dev, /dev/shm and /dev/pts (on Linux), and the paths configured with the sandbox-paths option. This is useful to prevent undeclared dependencies on files in directories such as /usr/bin. In addition, on Linux, builds run in private PID, mount, network, IPC and UTS namespaces to isolate them from other processes in the system (except that fixed-output derivations do not run in private network namespace to ensure they can access the network).
Currently, sandboxing only work on Linux and macOS. The use of a sandbox requires that Nix is run as root (so you should use the "build users" feature to perform the actual builds under different users than root).
If this option is set to relaxed, then fixed-output derivations and derivations that have the __noChroot attribute set to true do not run in sandboxes.
The default is true on Linux and false on all other platforms.

sandbox-dev-shm-size

This option determines the maximum size of the tmpfs filesystem mounted on /dev/shm in Linux sandboxes. For the format, see the description of the size option of tmpfs in mount(8). The default is 50%.

sandbox-paths

A list of paths bind-mounted into Nix sandbox environments. You can use the syntax *target*=*source* to mount a path in a different location in the sandbox; for instance, /bin=/nix-bin will mount the path /nix-bin as /bin inside the sandbox. If *source* is followed by ?, then it is not an error if *source* does not exist; for example, /dev/nvidiactl? specifies that /dev/nvidiactl will only be mounted in the sandbox if it exists in the host filesystem.
Depending on how Nix was built, the default value for this option may be empty or provide /bin/sh as a bind-mount of **bash**.

secret-key-files

A whitespace-separated list of files containing secret (private) keys. These are used to sign locally-built paths. They can be generated using **nix-store --generate-binary-cache-key**. The corresponding public key can be distributed to other users, who can add it to trusted-public-keys in their nix.conf.

show-trace

Causes Nix to print out a stack trace in case of Nix expression evaluation errors.

substitute

If set to `true` (default), Nix will use binary substitutes if available. This option can be disabled to force building from source.

stalled-download-timeout

The timeout (in seconds) for receiving data from servers during download. Nix cancels idle downloads after this timeout's duration.

substituters

A list of URLs of substituters, separated by whitespace. The default is `https://cache.nixos.org`.

system

This option specifies the canonical Nix system name of the current installation, such as `i686-linux` or `x86_64-darwin`. Nix can only build derivations whose `system` attribute equals the value specified here. In general, it never makes sense to modify this value from its default, since you can use it to 'lie' about the platform you are building on (e.g., perform a Mac OS build on a Linux machine; the result would obviously be wrong). It only makes sense if the Nix binaries can run on multiple platforms, e.g., 'universal binaries' that run on `x86_64-linux` and `i686-linux`.
It defaults to the canonical Nix system name detected by `configure` at build time.

system-features

A set of system "features" supported by this machine, e.g. `kvm`. Derivations can express a dependency on such features through the derivation attribute `requiredSystemFeatures`. For example, the attribute
requiredSystemFeatures = ["kvm"];
ensures that the derivation can only be built on a machine with the `kvm` feature.
This setting by default includes `kvm` if `/dev/kvm` is accessible, and the pseudo-features `nixos-test`, `benchmark` and `big-parallel` that are used in Nixpkgs to route builds to specific machines.

tarball-ttl

Default: `3600` seconds.
The number of seconds a downloaded tarball is considered fresh. If the cached tarball is stale, Nix will check whether it is still up to date using the ETag header. Nix will download a new version if the ETag header is unsupported, or the cached ETag doesn't match.
Setting the TTL to `0` forces Nix to always check if the tarball is up to date.
Nix caches tarballs in `$XDG_CACHE_HOME/nix/tarballs`.
Files fetched via `NIX_PATH`, `fetchGit`, `fetchMercurial`, `fetchTarball`, and `fetchurl` respect this TTL.

timeout

This option defines the maximum number of seconds that a builder can run. This is useful (for instance in an automated build system) to catch builds that are stuck in an infinite loop but keep writing to their standard output or standard error. It can be overridden using the `--timeout` command line switch.
The value `0` means that there is no timeout. This is also the default.

trace-function-calls

Default: `false`.
If set to `true`, the Nix evaluator will trace every function call. Nix will print a log message at the "vomit" level for every function entrance and function exit.
function-trace entered undefined position at 1565795816999559622
function-trace exited undefined position at 1565795816999581277
function-trace entered /nix/store/.../example.nix:226:41 at 1565795253249935150
function-trace exited /nix/store/.../example.nix:226:41 at 1565795253249941684
The `undefined position` means the function call is a builtin.
Use the `contrib/stack-collapse.py` script distributed with the Nix source code to convert the trace logs in to a format suitable for **flamegraph.pl**.

`trusted-public-keys`

A whitespace-separated list of public keys. When paths are copied from another Nix store (such as a binary cache), they must be signed with one of these keys. For example: `cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY= hydra.nixos.org-1:CNHJZBh9K4tP3EKF6FkkgeVYsS3ohTl+oS0Qa8bezVs=`.

`trusted-substituters`

A list of URLs of substituters, separated by whitespace. These are not used by default, but can be enabled by users of the Nix daemon by specifying `--option substituters` *`urls`* on the command line. Unprivileged users are only allowed to pass a subset of the URLs listed in `substituters` and `trusted-substituters`.

`trusted-users`

A list of names of users (separated by whitespace) that have additional rights when connecting to the Nix daemon, such as the ability to specify additional binary caches, or to import unsigned NARs. You can also specify groups by prefixing them with `@`; for instance, `@wheel` means all users in the `wheel` group. The default is `root`.